Annotated Bibliography for the Resilience Project

Rolf Riesen

May 12, 2009

Abstract

This document may serve as a resource for related work sections of papers produced by the resilience LDRD project.

Start

Not much here yet.

Notes

Just some random notes as I read papers...

- Many papers/systems concentrate on fault tolerance in grids and other wide-area distributed systems. This is usually done at the TCP layer.
- Fewer papers concentrate on very large scale parallel computing.

Glossary

Application driven checkpointing The application itself initiates a checkpoint and coordinates restart. Perhaps with some system-level support; e.g., telling the system what state needs to be saved and when.

Asynchronous checkpoint algorithm Each process takes checkpoints independently. This may lead to the *domino effect* during recovery. *Message logging* is a way to limit the domino effect.

Authenticated Byzantine fault An arbitrary or malicious fault, such as when one processor sends differing messages during a broadcast to its neighbors, that cannot imperceptibly alter an authenticated message. (From [Barborak et al. 1993])

BLCR Berkeley Lab's Linux Checkpoint/Restart [Duell et al. 2002a].

Byzantine fault Every fault possible in the system model. This fault class can be considered the universal fault set. (From [Barborak et al. 1993])

Coordinated checkpoint All processes in a system agree to save their local state to a global checkpoint at the same time. This guarantees coherence: no messages are in flight that would not be saved by any process. In case of a failure, all processes must roll back to the most recent successful checkpoint. Checkpoints can be taken in a blocking or non-blocking fashion (Coti, Herault, Lemarinier, Pilard, Rezmerita, Rodriguez, and Cappello 2006). Also see *synchronous checkpoint algorithm.*

Crash fault The fault that occurs when a processor loses its internal state or halts. For example, a PE that has had the contents of its instruction pipeline corrupted or has lost all power has suffered a crash fault. (From [Barborak et al. 1993])

CPR Checkpoint Restart

Domino effect If checkpoints are not coordinated, it is possible that a failed process needs to be restored back to a time before the last checkpoint of another process. This causes the second process to have to rollback to a yet earlier checkpoint, which in turn may force other processes to rollback as well [Johnson and Zwaenepoel 1988]. Message logging is a solution to this problem.

Fail-stop fault The fault that occurs when a processor ceases operation and alerts other processors of this fault [Schlichting and Schneider 1983]. (From [Barborak et al. 1993])

Incorrect computation fault The fault that occurs when a processor fails to produce the correct result in response to the correct inputs. (From [Barborak et al. 1993])

Message logging Processes save checkpoints independently from each other and when a process crashed, only that process gets recovered. To do so, requires that the external events since the last checkpoint of that process must be replayed. In systems of interest to us, those events are messages. That means either the sender or the receiver must log and save messages. Where to store messages and for how long is fairly complicated (compared to a coordinated checkpoint) and introduces overhead to message transmission.

Missing message When a process is rolled back it may expect a message which the sender would need to send a second time.

MTBF Mean Time Between Failures. This could be hardware or software and may, or may not, result in an interrupt of processing.

MTBI Mean Time Between Interrupts. Interruptions of processing caused by a failure; i.e., a job is aborted.

MTTF Mean Time To Failures. Same as MTBF(?)

MTTR Mean Time To Repair. Duration until a failed component can be used again.

Omission fault The fault that occurs when a processor fails to meet a deadline or begin a task. In particular, a send omission fault occurs when a processor fails to send a required message on time or at all, and a receive omission fault occurs when a processor fails to receive a required message and behaves as if it had not arrived. (From [Barborak et al. 1993])

Orphan message A message generated after a rollback to an earlier checkpoint which the receiver did receive earlier, but the sender does not remember due to the rollback.

Quasi-synchronous checkpoint algorithm Processes take checkpoints asynchronously interspersed with synchronized checkpoints at greater intervals.

RAS Reliability, availability, and serviceability.

Synchronous checkpoint algorithm Processes synchronize before taking a checkpoint. Also see *coordinated checkpoint*.

System driven checkpointing Automated (transparent) checkpointing. The system stores checkpoints of a running application automatically or with minimal assistance from the application.

Timing fault The fault that occurs when a processor completes a task either before or after its specified time frame or never. This is sometimes called a performance fault. (From [Barborak et al. 1993])

References

Ahn, J. (2007). **2-step algorithm for enhancing effectiveness of sender-based message logging**. In *SpringSim '07: Proceedings of the 2007 spring simulation multiconference*, San Diego, CA, USA, pp. 429–434. Society for Computer Simulation International. http://portal.acm.org/citation.cfm?id=1404748

ABSTRACT: Sender-based message logging allows each message to be logged in the volatile storage of its corresponding sender. This behavior avoids logging messages on the stable storage and results in lower failure-free overhead than receiver-based message logging. However, in the message logging approach, each process should keep in its limited volatile storage the log information of its sent messages for recovering their receivers. In this paper, we propose a 2-step algorithm to efficiently remove logged messages from the volatile storage while ensuring the consistent recovery of the system in case of process failures. As the first step, the algorithm eliminates useless log information in the volatile storage with no extra message and forced checkpoint. But, even if the step has been performed, the more empty buffer space for logging messages in future may be required. In this case, the second step forces the useful log information to become useless by maintaining a vector to record the size of the information for every other process. This behavior incurs fewer additional messages and forced checkpoints than existing algorithms.

COMMENT: This paper is hard to read because of its poor English. The idea behind message logging is to be able to replay messages since the last checkpoint. This allows failed processes to restart from the last checkpoint and catch up to the processes that did not fail. For this to work, messages must be logged by the sender or the receiver. Receiver logging has high overhead, since the messages must be sent to stable storage in case the receiver fails. Sender-based message logging is more efficient, but requires local, volatile storage. A protocol is needed to inform the sender when it no longer needs to retain old messages (garbage collection). An algorithm is also needed to send logged messages to stable storage when local storage on the send side becomes scarce. This paper describes such an algorithm. See also [Jiang and Manivannan 2007].

Barborak, M., A. Dahbura, and M. Malek (1993). **The consensus problem in fault-tolerant computing**. *ACM Comput. Surv. 25*(2), 171–220.

ABSTRACT: The consensus problem is concerned with the agreement on a system status by the fault-free segment of a processor population in spite of the possible inadvertent or even malicious spread of disinformation by the faulty segment of that population. The resulting protocols are useful throughout fault-tolerant parallel and distributed systems and will impact the design of decision systems to come. This paper surveys research on the consensus problem, compares approaches, outlines applications, and suggests directions for future work.

COMMENT: We are hoping that a RAS system will tell us when a component has failed and we need to do something to recover. This paper explains what the RAS system must do, or ourselves in some situations. It is also a nice paper in the sense that it gives clear definitions for the different types of faults. We need these when we describe what our system can, or cannot, handle.

DOI: http://doi.acm.org/10.1145/152610.152612

Bartlett, J. F. (1981). **A NonStop kernel**. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*, New York, NY, USA, pp. 22–29. ACM.

ABSTRACT: The Tandem NonStop System is a fault-tolerant [1], expandable, and distributed computer system designed expressly for online transaction processing. This paper describes the key primitives of the kernel of the operating system. The first section describes the basic hardware building blocks and introduces their software analogs: processes

and messages. Using these primitives, a mechanism that allows fault-tolerant resource access, the process-pair, is described. The paper concludes with some observations on this type of system structure and on actual use of the system.

COMMENT: Describes the hardware and software of the Tandem system. The hardware helps ensure the fail-stop model and provides mechanisms for fault detection. The system software makes use of these machine characteristics and enables error recovery. Processes are duplicated and mechanisms are in place for the primary process to carry out externally visible events. I/O for example or messages to other processes. The backup process receives all the necessary information from the primary process so it can take over if necessary. The system assists in making sure that requests are retried if necessary but not duplicated. It seems users must explicitly write these process pairs. However, applications written in COBOL do not have (cannot have?) backup processes.

DOI: http://doi.acm.org/10.1145/800216.806587

Blough, D. M. and P. Liu (2000). **FIMD-MPI: A Tool for Injecting Faults into MPI Applications**. *Parallel and Distributed Processing Symposium, International 0*, 241.

ABSTRACT: Parallel computing is seeing increasing use in critical applications. The need therefore arises to test the robustness of parallel applications in the presence of exceptional conditions, or faults. Communication-software-based fault injection is an extremely flexible approach to robustness testing in message-passing parallel computers. A fault injection methodology and tool that use this approach are presented. The tool, known as FIMD-MPI, allows injection of faults into MPI-based applications. The structure and operation of FIMD-MPI are described and the use of the tool is illustrated on an example fault-tolerant MPI application

COMMENT: An infrastructure that can inject various types of faults. An application must be recompiled to use this infrastructure, although they could probably have used the MPI profiling interface. A configuration file specifies the types and number of faults to inject at run time.

DOI: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2000.845991

Bronevetsky, G., R. Fernandes, D. Marques, K. Pingali, and P. Stodghill (2006, April). **Recent advances in checkpoint/recovery systems**.

ABSTRACT: Checkpoint and recovery (CPR) systems have many uses in high-performance computing. Because of this, many developers have implemented it, by hand, into their applications. One of the uses of check-pointing is to help mitigate the effects of interruptions in computational service (both planned and unplanned) In fact, some supercomputing centers expect their users to use checkpointing as a matter of policy. And yet, few centers provide fully automatic checkpointing systems for their high-end production machines. The paper is a status report on our work on the family of C3 systems for (almost) fully automatic checkpointing for scientific applications. To date, we have shown that our techniques can be used for checkpointing sequential, MPI and OpenMP applications written inC, Fortran, and several other languages. A novel aspect of our work is that we have not built a single checkpointing system, rather, we have developed a methodology and a set of techniques that have enabled us to develop a number of systems, each meeting different design goals and efficiency requirements

COMMENT: Referenced by Maloney and Goscinski (2009) for the examples of rand() and MPI to explain that both system-level and application-level CPR are needed to assure correctness. The seed is not exposed to the application by the POSIX API. It can be set, but not read for check-pointing. In MPI, the rank to physical node ID (e.g., IP addresses) maps have no meaning to the system (kernel). Upon recovery they would be restored, even though new maps may be needed.

DOI: http://dx.doi.org/10.1109/IPDPS.2006.1639575

Coti, C., T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello (2006). **Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI** . In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, pp. 127. ACM.

ABSTRACT: A long-term trend in high-performance computing is the increasing number of nodes in parallel computing platforms, which entails a higher failure probability. Fault tolerant programming environments should be used to guarantee the safe execution of critical applications. Research in fault tolerant MPI has led to the development of several fault tolerant MPI environments. Different approaches are being proposed using a variety of fault tolerant message passing protocols based on coordinated checkpointing or message logging. The most popular approach is with coordinated checkpointing. In the literature, two different concepts of coordinated checkpointing have been proposed: blocking and nonblocking. However they have never been compared quantitatively and their respective scalability remains unknown. The contribution of this paper is to provide the first comparison between these two approaches and a study of their scalability. We have implemented the two approaches within the MPICH environments and evaluate their performance using the NAS parallel benchmarks.

COMMENT: Compares two implementations of coordinated checkpointing inside MPICH. One is blocking, the other is not. The implementation is over TCP and uses TCP's keepalive parameter for failure detection. Detection latency is on the order of minutes, except the authors kill tasks to inject faults. The OS (Linux) survives and notifies the other end of the channel immediately. To write the actual checkpoint, BLCR [Duell et al. 2002a] is used. The paper shows NAS parallel benchmark results and shows that, as the checkpoint interval decreases, non-blocking checkpoints add less overhead. This is especially true for lower performing networks. According to the paper, in high-performance networks, blocking checkpoints might be OK for *"sensible checkpoint frequency"*. What seems to be missing is the cost of roll-back when that has to be done frequently as we expect in the near future. They did perform some experiments on clusters but seem mostly interested in grids. Even when running over Myrinet they were using the TCP layer.

DOI: http://doi.acm.org/10.1145/1188455.1188587

Duell, J., P. Hargrove, and E. Roman (2002a, December). **The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart**. Technical Report LBNL-54941, Berkeley Lab. https://ftg.lbl.gov/CheckpointRestart/Pubs/blcr.pdf

ABSTRACT: *No abstract.*

COMMENT: This is the tech report that describes the original version of BLCR. [Hargrove and Duell 2006] is a much more recent version but contains no details.

Duell, J., P. Hargrove, and E. Roman (2002b, May). **Requirements for Linux Checkpoint/Restart**. Technical Report LBNL-49659, Berkeley Lab. https://ftg.lbl.gov/CheckpointRestart/Pubs/LBNL-49659.pdf

ABSTRACT: *No abstract.*

COMMENT: This is a formal description of the requirements for BLCR. It does not discuss whether the implementation meets these requirements or not. There is no mention whether these requirements are sufficient.

Elnozahy, E. and J. Plank (2004, April). **Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery**. *Dependable and Secure Computing, IEEE Transactions on 1*(2), 97–108.

ABSTRACT: Over the past two decades, rollback-recovery via checkpoint-restart has been used with reasonable success for long-running applications, such as scientific workloads that take from few hours to few months to complete. Currently, several commercial systems and publicly available libraries exist to support various flavors of checkpointing. Programmers typically use these systems if they are satisfactory or otherwise embed checkpointing support themselves within the application. In this paper, we project the performance and functionality of checkpointing algorithms and systems as we know them today into the future. We start by surveying the current technology roadmap and particularly how Peta-Flop capable systems may be plausibly constructed in the next few years. We consider how rollback-recovery as practiced today will fare when systems may have to be constructed out of thousands of nodes. Our projections predict that, unlike current practice, the effect of rollback-recovery may play a more prominent role in how systems may be configured to reach the desired performance level. System planners may have to devote additional resources to enable rollback-recovery and the current practice of using ”cheap commodity” systems to form large-scale clusters may face serious obstacles. We suggest new avenues for research to react to these trends.

COMMENT: This paper makes a clear case for the need to come up with something smarter than simple check-point restart in 100,000 and more processor systems. Somewhat similar to [Oldfield et al. 2007], but it does not really make any suggestions on how to fix the problem. The paper has a graph that shows how checkpointing overhead impacts speedup. It also shows that improving the time to do a checkpoint helps some, but if many recoveries are expected that the cost of these recoveries limit the scalability of an application.

DOI: http://dx.doi.org/10.1109/TDSC.2004.15

Elnozahy, E. N. M., L. Alvisi, Y.-M. Wang, and D. B. Johnson (2002). **A survey of rollback-recovery protocols in message-passing systems**. *ACM Comput. Surv. 34*(3), 375–408.

ABSTRACT: This survey covers rollback-recovery techniques that do not require special language constructs. In the first part of the survey we classify rollback-recovery protocols into checkpoint-based and log-based. Checkpoint-based protocols rely solely on checkpointing for system state restoration. Checkpointing can be coordinated, uncoordinated, or communication-induced. Log-based protocols combine checkpointing with logging of nondeterministic events, encoded in tuples called determinants. Depending on how determinants are logged, log-based protocols can be pessimistic, optimistic, or causal. Throughout the survey, we highlight the research issues that are at the core of rollback-recovery and present the solutions that currently address them. We also compare the performance of different rollback-recovery protocols with respect to a series of desirable properties and discuss the issues that arise in the practical implementations of these protocols.

COMMENT: Reviews checkpoint-based and log-based rollback-recovery techniques. It nicely explains the reasons for each technique, its drawbacks and advantages.

DOI: http://doi.acm.org/10.1145/568522.568525

Florio, V. D. and C. Blondia (2008). **A survey of linguistic structures for application-level fault tolerance**. *ACM Comput. Surv. 40*(2), 1–37.

ABSTRACT: Structures for the expression of fault-tolerance provisions in application software comprise the central topic of this article. Structuring techniques answer questions as to how to incorporate fault tolerance in the application layer of a computer program and how to manage the fault-tolerant code. As such, they provide the means to control complexity, the latter being a relevant factor for the introduction of design faults. This fact and the ever-increasing complexity of today’s distributed software justify the need for simple, coherent, and effective structures for the expression of fault-tolerance in the application software. In this text we first define a “base” of structural attributes with which application-level fault-tolerance structures can be qualitatively assessed and compared with each other and with respect to the aforementioned needs. This result is then used to provide an elaborated survey of the state-of-the-art of application-level fault-tolerance structures.

COMMENT: This paper argues for software fault-tolerance; i.e., it says that hardware alone cannot solve the problem of making applications survive faults in the system. The reason for this is the increasing complexity of software in these systems and the likelihood that software fails. The paper then goes on to survey the techniques that enable software fault-tolerance. I need to read this more carefully to see if we can apply some of these techniques in libraries or at the system level.

DOI: `http://doi.acm.org/10.1145/1348246.1348249`

Gärtner, F. C. (1999). **Fundamentals of fault-tolerant distributed computing in asynchronous environments**. *ACM Comput. Surv. 31*(1), 1–26.

ABSTRACT: Fault tolerance in distributed computing is a wide area with a significant body of literature that is vastly diverse in methodology and terminology. This paper aims at structuring the area and thus guiding readers into this interesting field. We use a formal approach to define important terms like fault, fault tolerance, and redundancy. This leads to four distinct forms of fault tolerance and to two main phases in achieving them: detection and correction. We show that this can help to reveal inherently fundamental structures that contribute to understanding and unifying methods and terminology. By doing this, we survey many existing methodologies and discuss their relations. The underlying system model is the close-to-reality asynchronous message-passing model of distributed computing.

COMMENT: This is a very formal definition of fault models and fault tolerance; much more formal than [Barborak et al. 1993].

DOI: `http://doi.acm.org/10.1145/311531.311532`

Hargrove, P. H. and J. C. Duell (2006, June). **Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters**. Technical Report LBNL-60520, Berkeley Lab. `https://ftg.lbl.gov/CheckpointRestart/Pubs/LBNL-60520.pdf`

ABSTRACT: This article describes the motivation, design and implementation of Berkeley Lab Checkpoint/Restart (BLCR), a system-level checkpoint/restart implementation for Linux clusters that targets the space of typical High Performance Computing applications, including MPI. Application-level solutions, including both checkpointing and fault-tolerant algorithms, are recognized as more time and space efficient than system-level checkpoints, which cannot make use of any application-specific knowledge. However, system-level checkpointing allows for preemption, making it suitable for responding to fault precursors (for instance, elevated error rates from ECC memory or network CRCs, or elevated temperature from sensors). Preemption can also increase the efficiency of batch scheduling; for instance reducing idle cycles (by allowing for shutdown without any queue draining period or reallocation of resources to eliminate idle nodes when better fitting jobs are queued), and reducing the average queued time (by limiting large jobs to running during off-peak hours, without the need to limit the length of such jobs). Each of these potential uses makes BLCR a valuable tool for efficient resource management in Linux clusters.

COMMENT: A very brief description of the goals and design of BLCR. The original tech report [Duell et al. 2002a] has more information than this paper.

Jiang, Q., Y. Luo, and D. Manivannan (2008). **An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems**. *J. Parallel Distrib. Comput. 68*(12), 1575–1589.

ABSTRACT: Checkpointing and rollback recovery are widely used techniques for achieving fault-tolerance in distributed systems. In this paper, we present a novel checkpointing algorithm which has the following desirable features: A process can independently initiate consistent global checkpointing by saving its current state, called a tentative checkpoint. Other

processes come to know about a consistent global checkpoint initiation through information piggy-backed with the application messages or limited control messages if necessary. When a process comes to know about a new consistent global checkpoint initiation, it takes a tentative checkpoint after processing the message (not before processing the message as in existing communication-induced checkpointing algorithms). After a process takes a tentative checkpoint, it starts logging the messages sent and received in memory. When a process comes to know that every other process has taken a tentative checkpoint corresponding to current consistent global checkpoint initiation, it flushes the tentative checkpoint and the message log to the stable storage. The tentative checkpoints together with the message logs stored in the stable storage form a consistent global checkpoint. Two or more processes can concurrently initiate consistent global checkpointing by taking a new tentative checkpoint; in that case, the tentative checkpoints taken by all these processes will be part of the same consistent global checkpoint. The sequence numbers assigned to checkpoints by a process increase monotonically. Checkpoints with the same sequence number form a consistent global checkpoint. We also present the performance evaluation of our algorithm.

COMMENT: This is the journal version of [Jiang and Manivannan 2007]. It contains a performance evaluation of their algorithm.

DOI: `http://dx.doi.org/10.1016/j.jpdc.2008.08.003`

Jiang, Q. and D. Manivannan (2007, March). **An optimistic checkpointing and selective message logging approach for consistent global checkpoint collection in distributed systems**.

ABSTRACT: In this paper, we present an asynchronous consistent global checkpoint collection algorithm which prevents contention for network storage at the file server and hence reduces the checkpointing overhead. The algorithm has two phases: In the first phase, a process initiates consistent global checkpoint collection by saving its state tentatively and asynchronously (called tentative checkpoint) in local memory or remote stable storage if there is no contention for stable storage while saving the state; in the second phase, the message log associated with the tentative checkpoint is stored in stable storage (checkpoint finalization phase). The tentative checkpoint together with the associated message log stored in the stable storage becomes part of a consistent global checkpoint. Under our algorithm, two or more processes can concurrently initiate consistent global checkpoint collection. Every tentative checkpoint will be finalized successfully unless a failure occurs. The finalized checkpoints of each process is assigned a unique sequence number in ascending order. Finalized checkpoints with same sequence number form a consistent global checkpoint.

COMMENT: This looks similar to [Ahn 2007]. but is much more readable. It is not clear to me how (and whether) the two algorithms differ, due to the difficulty of reading [Ahn 2007]. This paper argues for uncoordinated checkpoints to ease contention at the I/O system. For performance reasons their algorithm also takes regular, coordinated checkpoints. The journal version of this article is [Jiang et al. 2008]

DOI: `http://dx.doi.org/10.1109/IPDPS.2007.370308`

Johnson, D. B. and W. Zwaenepoel (1988). **Recovery in distributed systems using asynchronous message logging and checkpointing**. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, New York, NY, USA, pp. 171–181. ACM.

ABSTRACT: In a distributed system using message logging and checkpointing to provide fault tolerance, there is always a unique maximum recoverable system state, regardless of the message logging protocol used. The proof of this relies on the observation that the set of system states that have occurred during any single execution of a system forms a lattice, with the sets of consistent and recoverable system states as sublattices. The maximum recoverable system state never decreases, and if all messages are eventually logged, the domino effect cannot occur. This paper presents a general model for reasoning about recovery in such a system and, based on this model, an efficient algorithm for determining the maximum recoverable system state at any time. This work unifies existing approaches to fault tolerance based on message logging and checkpointing, and improves on existing methods for optimistic recovery in distributed systems.

COMMENT: This paper describes (formally) how message logging works and how it can prevent the domino effect.

DOI: http://doi.acm.org/10.1145/62546.62575

Jung, H., D. Shin, H. Han, J. W. Kim, H. Y. Yeom, and J. Lee (2005). **Design and Implementation of Multiple Fault-Tolerant MPI over Myrinet (M^3)**. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, pp. 32. IEEE Computer Society.

ABSTRACT: Advances in network technology and computing power have inspired the emergence of high-performance cluster computing systems. While cluster management and hardware highavailability tools are readily available, practical and easily deployable fault-tolerant systems have not been successfully adopted commercially. We present a fault-tolerant system, Multiple fault-tolerant MPI over Myrinet (M3), that differs in notable respects from other proposed fault-tolerant systems in the literature. M3 is built on top of Myrinet since it is regarded as one of the best solutions for highperformance networks and is widely used in cluster computing systems because it can provide a high-speed switching network that is an inevitable ingredient in interconnecting clusters of workstations or PCs. M^3 is a user-transparent checkpointing system for multiple fault-tolerant MPI implementation that is primarily based on the coordinated checkpointing protocol. M3 supports three critical functionalities that are necessary for faulttolerance: a light-weight failure detection mechanism, dynamic process management that includes process migration, and a consistent checkpoint and recovery mechanism. The features of M are that it requires no modifications of application code and that it preserves much of the high performance characteristics of Myrinet. This paper describes the architecture of M3, its detailed design principles and comprehensive implementation issues. We also propose practical solutions for those involved in constructing highly available cluster systems for parallel programming systems. Experimental results substantiate our assertion that M3 can be a good candidate for practically deployable fault-tolerant systems in very-large and high-performance Myrinet clusters and that its protocol can be applied to a wide variety of parallel communication libraries without difficulty.

COMMENT: Describes a user-transparent fault detection and recovery system over Myrinet. Their earlier work was done at the TCP layer, while in this paper they describe a mechanism at the ADI/GM layer of MPICH. One of the difficulties they deal with are eager sends and zero-copy DMAs, since faults may occur in the middle of these protocols.

DOI: http://dx.doi.org/10.1109/SC.2005.22

Li, K., J. F. Naughton, and J. S. Planck (1991, September). **Checkpointing multicomputer applications**. *Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on*, 2–11.

ABSTRACT: The authors present a checkpointing scheme that is transparent, imposes overhead only during checkpoints, requires minimal message logging, and allows for quick resumption of execution from a checkpointed image. Since checkpointing multicomputer applications poses requirements different from those posed by checkpointing general distributed systems, existing distributed checkpointing schemes are inadequate for multicomputer checkpointing. The proposed checkpointing scheme makes use of special properties of multicomputer interconnection networks to satisfy this set of requirements. The proposed algorithm is efficient both when taking checkpoints and when recovering from checkpointed images

COMMENT: This paper tries to distinguish checkpoint/restart in a distributed (grid) system from doing it in a parallel system. It cites performance, less autonomous nodes, and deterministic deadlock-free algorithms for needing a solution specific to these systems. Their

algorithm depends on point-to-point message ordering characteristics of parallel systems and avoids the $O(n^2)$ number of messages to coordinate the next checkpoint.

DOI: http://dx.doi.org/10.1109/RELDIS.1991.145398

Maloney, A. and A. Goscinski (2009, April). **A survey and review of the current state of rollback-recovery for cluster systems**. *Concurrency and Computation: Practice and Experience*.

ABSTRACT: A variety of research problems exist that require considerable time and computational resources to solve. Attempting to solve these problems produces long-running applications that require a reliable and trustworthy system upon which they can be executed. Cluster systems provide an excellent environment upon which to run these applications because of their low cost to performance ratio; however, due to being created using commodity components they are prone to failures. This report surveyed and reviewed the issues currently relating to providing fault tolerance for long-running applications. Several fault tolerance approaches were investigated; however, it was found that rollback-recovery provides a favourable approach for user applications in cluster systems. Two facilities are required to provide fault tolerance using rollback-recovery: checkpointing and recovery. It was shown here that a multitude of work has been done for enhancing checkpointing; however, the intricacies of providing recovery have been neglected. The problems associated with providing recovery include; providing transparent and autonomic recovery, selecting appropriate recovery computers, and maintaining a consistent observable behaviour when an application fails.

COMMENT: A survey of seven specific rollback-recovery systems for clusters. This is in contrast to the more thorough and academic [Elnozahy et al. 2002]. I.e., this paper is more like a product overview/review, while Elnozahy et al. (2002) review techniques.

DOI: http://dx.doi.org/10.1002/cpe.1413

Mandal, P. S. and K. Mukhopadhyaya (2006). **Performance analysis of different checkpointing and recovery schemes using stochastic model**. *Journal of Parallel and Distributed Computing 66*(1), 99 – 107.

ABSTRACT: Several schemes for checkpointing and rollback recovery have been reported in the literature. In this paper, we analyze some of these schemes under a stochastic model. We have derived expressions for average cost of checkpointing, rollback recovery, message logging and piggybacking with application messages in synchronous as well as asynchronous checkpointing. For quasi-synchronous checkpointing we show that in a system with n processes, the upper bound and lower bound of selective message logging are O(n2) and O(n), respectively.

COMMENT: Nicely introduces the concepts of synchronous and asynchronous checkpoint algorithms. The authors create models for the various checkpoint scenarios, including the cost of recovery. The recovery cost of message logging is dependent on the message rate. The overhead of taking a checkpoint is highest for synchronous algorithms, and lowest for asynchronous algorithms. Quasi-synchronous is in-between.

DOI: http://dx.doi.org/10.1016/j.jpdc.2005.06.013

Nightingale, E. B., K. Veeraraghavan, P. M. Chen, and J. Flinn (2008). **Rethink the sync**. *ACM Trans. Comput. Syst. 26*(3), 1–26.

ABSTRACT: We introduce external synchrony, a new model for local file I/O that provides the reliability and simplicity of synchronous I/O, yet also closely approximates the performance of asynchronous I/O. An external observer cannot distinguish the output of a computer with an externally synchronous file system from the output of a computer with a synchronous file system. No application modification is required to use an externally synchronous file system. In fact, application developers can program to the simpler synchronous I/O abstraction and still receive excellent performance. We have implemented an externally synchronous file system for Linux, called xsyncfs. Xsyncfs provides the same durability and ordering-guarantees as those provided by a synchronously mounted ext3 file system. Yet even for I/O-intensive benchmarks, xsyncfs performance is within 7%; of ext3 mounted asynchronously. Compared to ext3 mounted synchronously, xsyncfs is up to two orders of magnitude faster.

COMMENT: Probably not directly relevant. It uses fault tolerance techniques to improve file system performance.

DOI: http://doi.acm.org/10.1145/1394441.1394442

Oldfield, R. A., S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth (2007, September). **Modeling the Impact of Checkpoints on Next-Generation Systems**. In *24th IEEE Conference on Mass Storage Systems and Technologies*, pp. 30–46. IEEE Computer Society.

ABSTRACT: The next generation of capability-class, massively parallel processing (MPP) systems is expected to have hundreds of thousands of processors. For application-driven, periodic checkpoint operations, the state-of-the-art does not provide a solution that scales to next-generation systems. We demonstrate this by using mathematical modeling to compute a lower bound of the impact of these approaches on the performance of applications executed on three massive-scale, in-production, DOE systems and a theoretical petaflop system. We also adapt the model to investigate a proposed optimization that makes use of "lightweight" storage architectures and overlay networks to overcome the storage system bottleneck. Our results indicate that (1) as we approach the scale of next-generation systems, traditional checkpoint/restart approaches will increasingly impact application performance, accounting for over 50% of total application execution time; (2) although our alternative approach improves performance, it has limitations of its own; and (3) there is a critical need for new approaches to checkpoint/restart that allow continuous computing with minimal impact on the scalability of applications.

COMMENT: The paper first justifies the need to improve checkpoint/restart. It then goes on to show how lightweight file systems and overlay nodes can improve the situation. The authors modify Daly's formula to compute the optimal checkpoint interval so it takes the overlay nodes into account.

DOI: http://dx.doi.org/10.1109/MSST.2007.24

Oliner, A. J., R. K. Sahoo, J. E. Moreira, and M. Gupta (2005). **Performance Implications of Periodic Checkpointing on Large-Scale Cluster Systems**. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, Washington, DC, USA, pp. 299.2. IEEE Computer Society.

ABSTRACT: Large-scale systems like BlueGene/L are susceptible to a number of software and hardware failures that can affect system performance. Periodic application checkpointing is a common technique for mitigating the amount of work lost due to job failures, but its effectiveness under realistic circumstances has not been studied. In this paper, we analyze the system-level performance of periodic application checkpointing using parameters similar to those projected for BlueGene/L systems. Our results reflect simulations on a toroidal interconnect architecture, using a real job log from a machine similar to BlueGene/L, and with a real failure distribution from a large-scale cluster. Our simulation studies investigate the impact of parameters such as checkpoint overhead and checkpoint interval on a number of performance metrics, including bounded slowdown, system utilization, and total work lost. The results suggest that periodic checkpointing may not be an effective way to improve the average bounded slowdown or average system utilization metrics, though it reduces the amount of work lost due to failures. We show that overzealous checkpointing with high overhead can amplify the effects of failures. The study also suggests that new metrics and checkpointing techniques may be required to effectively handle job failures on large-scale machines like BlueGene/L.

COMMENT: The authors use a from a Cray T3D and a failure log from an AIX cluster to simulate checkpointing and recovery of a BlueGene/L machine. Among other conclusions, they say that periodic checkpointing may not be the best choice since failures are not independent and not identically distributed.

DOI: http://dx.doi.org/10.1109/IPDPS.2005.337

Raynal, M. (2001). **Quiescent Uniform Reliable Broadcast as an Introduction to Failure Detector Oracles**. In *PaCT '01: Proceedings of the 6th International Conference on Parallel Computing Technologies*, London, UK, pp. 98–111. Springer-Verlag.

ABSTRACT: This paper is a short and informal introduction to failure detector oracles for asynchronous distributed systems prone to process crashes and fair lossy channels. A distributed coordination problem (the implementation of Uniform Reliable Broadcast with a quiescent protocol) is used as a paradigm to visit two types of such oracles. One of them is a "guessing" oracle in the sense that it provides a process with information that the processes could only approximate if they had to compute it. The other is a "hiding" oracle in the sense that it allows to isolate and encapsulate the part of a protocol that has not the required behavioral properties. A quiescent uniform reliable broadcast protocol is described. The guessing oracle is used to ensure the "uniformity" requirement stated in the problem specification. The hiding oracle is used to ensure the additional "quiescence" property that the protocol behavior has to satisfy.

DOI: http://dx.doi.org/10.1007/3-540-44743-1_10

Saito, Y. and M. Shapiro (2005). **Optimistic replication**. *ACM Comput. Surv. 37*(1), 42–81.

ABSTRACT: Data replication is a key technology in distributed systems that enables higher availability and performance. This article surveys optimistic replication algorithms. They allow replica contents to diverge in the short term to support concurrent work practices and tolerate failures in low-quality communication links. The importance of such techniques is increasing as collaboration through wide-area and mobile networks becomes popular. Optimistic replication deploys algorithms not seen in traditional "pessimistic" systems. Instead of synchronous replica coordination, an optimistic algorithm propagates changes in the background, discovers conflicts after they happen, and reaches agreement on the final contents incrementally. We explore the solution space for optimistic replication algorithms. This article identifies key challenges facing optimistic replication systems – ordering operations, detecting and resolving conflicts, propagating changes efficiently, and bounding replica divergence – and provides a comprehensive survey of techniques developed for addressing these challenges.

COMMENT: Aimed more at distributed systems.

DOI: http://doi.acm.org/10.1145/1057977.1057980

Schlichting, R. D. and F. B. Schneider (1983). **Fail-stop processors: an approach to designing fault-tolerant computing systems**. *ACM Trans. Comput. Syst. 1*(3), 222–238.

ABSTRACT: A methodology that facilitates the design of fault-tolerant computing systems is presented. It is based on the notion of a fail-stop processor. Such a processor automatically halts in response to any internal failure and does so before the effects of that failure become visible. The problem of implementing processors that, with high probability, behave like fail-stop processors is addressed. Axiomatic program verification techniques are described for use in developing provably correct programs for fail-stop processors. The design of a process control system illustrates the use of our methodology.

COMMENT: Describes the fail-stop model that is assumed by most checkpoint/restart algorithms.

DOI: http://doi.acm.org/10.1145/357369.357371

Index

A

A

abort . . . 2
application driven checkpointing . . . [1]
asynchronous checkpoint algorithm . . . [1], 10
authenticated Byzantine fault . . . [1]
authors
- Ahn . . . 8
- Ahn, Jinho . . . (3)
- Alvisi, Lorenzo . . . (6)
- Arunagiri, Sarala . . . (11)
- Barborak et al. . . . 1, 2, 7
- Barborak, Michael . . . (3)
- Bartlett, Joel F. . . . (3)
- Blondia, Chris . . . (6)
- Blough, Douglas M. . . . (4)
- Bronevetsky, G. . . . (4)
- Cappello, Franck . . . (5)
- Chen, Peter M. . . . (10)
- Coti, Camille . . . (5)
- Dahbura, Anton . . . (3)
- Duell et al. . . . 1, 5, 7
- Duell, Jason . . . (5)
- Duell, Jason C. . . . (7)
- Elnozahy et al. . . . 10
- Elnozahy, E. N. (Mootaz) . . . (6)
- Elnozahy, E.N. . . . (5)
- Fernandes, R. . . . (4)
- Flinn, Jason . . . (10)
- Florio, Vincenzo De . . . (6)
- Gärtner, Felix C. . . . (7)
- Goscinski, Andrzej . . . (10)
- Gupta, M. . . . (11)
- Han, Hyuck . . . (9)
- Hargrove and Duell . . . 5
- Hargrove, Paul . . . (5)
- Hargrove, Paul H. . . . (7)
- Herault, Thomas . . . (5)
- Jiang and Manivannan . . . 3, 8
- Jiang et al. . . . 8
- Jiang, Q. . . . (8)
- Jiang, Qiangfeng . . . (7)
- Johnson and Zwaenepoel . . . 2
- Johnson, David B. . . . (6), (8)
- Jung, Hyungsoo . . . (9)
- Kim, Jai W. . . . (9)
- Lee, Jongsuk . . . (9)
- Lemarinier, Pierre . . . (5)
- Li, Kai . . . (9)
- Liu, Peng . . . (4)
- Luo, Yi . . . (7)
- Malek, Miroslaw . . . (3)
- Maloney, Andrew . . . (10)
- Mandal, Partha Sarathi . . . (10)
- Manivannan, D. . . . (7, 8)
- Marques, D. . . . (4)
- Moreira, J. E. . . . (11)
- Mukhopadhyaya, Krishnendu . . . (10)
- Naughton, Jeffrey F. . . . (9)
- Nightingale, Edmund B. . . . (10)
- Oldfield et al. . . . 6
- Oldfield, Ron A. . . . (11)
- Oliner, A. J. . . . (11)
- Pilard, Laurence . . . (5)
- Pingali, K. . . . (4)
- Planck, James S. . . . (9)
- Plank, J.S. . . . (5)
- Raynal, Michel . . . (12)
- Rezmerita, Ala . . . (5)
- Riesen, Rolf . . . (11)
- Rodriguez, Eric . . . (5)
- Roman, Eric . . . (5)
- Roth, Philip C. . . . (11)
- Sahoo, R. K. . . . (11)
- Saito, Yasushi . . . (12)
- Schlichting and Schneider . . . 2
- Schlichting, Richard D. . . . (12)
- Schneider, Fred B. . . . (12)
- Seelam, Seetharami . . . (11)
- Shapiro, Marc . . . (12)
- Shin, Dongin . . . (9)
- Stodghill, P. . . . (4)
- Teller, Patricia J. . . . (11)
- Varela, Maria Ruiz . . . (11)
- Veeraraghavan, Kaushik . . . (10)
- Wang, Yi-Min . . . (6)
- Yeom, Heon Y. . . . (9)
- Zwaenepoel, Willy . . . (8)

automated checkpoint . . . 9
automated checkpointing . . . [2]

B

BLCR . . . [1], 5, 7
blocking checkpoint . . . 1, 5
BlueGene/L . . . 11
Byzantine fault . . . [1]

C

C . . . 4
checkpoint

C

automated .... *see* automated checkpoint
blocking .... *see* blocking checkpoint
coordinated .... [1], 2, 5, 6, 9
forced 3
independent 6
log-based 6
non-blocking .... *see* non-blocking checkpoint
tentative 8
transparent .... *see* automated checkpoint
uncoordinated .... *see* independent

checkpoint algorithm
asynchronous .... *see* asynchronous checkpoint algorithm
quasi-synchronous ..... *see* quasi-synchronous checkpoint algorithm
synchronous ...... *see* synchronous checkpoint algorithm

COBOL 4
consensus 3
Coordinated checkpoint 2
CPR .... [2]
crash fault .... [1]
Cray T3D 12

D

domino effect .... 1, [2], 8, 9

E

eager sends 9

F

fail-stop fault .... [2]
fail-stop model .... [2], 4, 12
failure 2
fault (type)
authenticated Byzantine .... [1]
Byzantine .... [1]
crash .... [1]
fail-stop .... [2]
incorrect computation .... [2]
omission .... [2]
timing .... [2]

fault injection 4, 5
fault-tolerant 3–7, 9, 10
formal definition 7

Fortran 4

I

incorrect computation fault .... [2]

J

job log 12

M

message logging .... 1, [2], 2, 3, 5, 8–10
missing message .... [2]
MPI 4, 5, 9
MPICH 5
MTBF .... [2]
MTBI .... [2]
MTTF .... [2]
MTTR .... [2]
Myrinet 5, 9

N

non-blocking checkpoint 1, 5
NonStop 3

O

omission fault .... [2]
OpenMP 4
orphan message .... [2]

Q

quasi-synchronous checkpoint algorithm .... [2]

R

RAS .... [2], 3
redundancy 7
replication 12

S

stable storage 3, 8
synchronous checkpoint algorithm .... [2], 10
system driven checkpointing .... [2]

T

Tandem 3
TCP layer 1, 5, 9
tentative checkpoint 8
timing fault .... [2]

U

user-transparent .... *see* automated checkpoint

(*n*) page *n* is in the bibliography.
[*n*] page *n* is in the glossary.
n page of a definition or a main entry.
n other pages where an entry is mentioned.